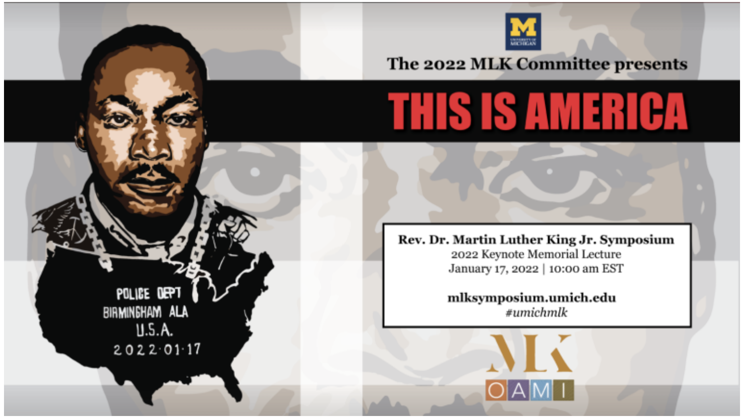
EECS 280 - Lecture 3

Pointers

https://eecs280staff.github.io/notes/03_Pointers.html

Announcements

- P1 due 1/19 (little over a week)
  - Will discuss a bit today
- Remember to fill stuff out!
  - CARES survey (0.5% of your total grade) by 1/26
  - Coaching form (Piazza @105)
  - Exam accommodations form
  - Exam conflict forms 1/28
- Labs start this week
- Office hours in full swing – see website
- No lecture / OH on Monday



Last Time

- Functions
  - What the call stack is, and how it works
- Procedural abstraction
  - Why we use so many files, and how to use them properly
  - `#include` .h files, pass .cpp files to compiler (setup in your IDE)

Today

- How to write effective tests

- Pointers
  - What they are and how to use them

Checking the REQUIRES Clause?

```
// REQUIRES: v is not empty
// EFFECTS: returns median of the numbers in v
double median(std::vector<double> v) {
  if (v.empty()) {
    // try to salvage the situation
  }
}
```

Don't do this.

```
// REQUIRES: v is not empty
// EFFECTS: returns median of the numbers in v
double median(std::vector<double> v) {
  assert(!v.empty()); // sound the alarms!
}
```

Do this.

assert([EXPRESSION])

- assert() is a programmer's friend for debugging
- Does nothing if EXPRESSION is true
- Exits and prints an error message if EXPRESSION is false

```
#include <cassert>
int main() {
  int x = 3;
  int y = 4;
  assert(x < y); // ok, does nothing
  assert(x > y); // crash with debug message
}
```

```
$ ./test
Assertion failed: (false), function main,
file test.cpp, line 6.
```

Important!

Triple check that the REQUIRES clause so you aren't being overly restrictive!

Properties of Procedural Abstraction

- Local
  The implementation of an abstraction can be understood without examining any other abstraction implementation.

- Substitutable
  You can replace one (correct) implementation of an abstraction with another (correct) one, without having to change the way the abstraction is used.

  Separation of interface from implementation:
  Only depend on interface, not implementation!

Substitutability Example

- Here's the current implementation in p1_library.cpp:

```
void sort(std::vector<double> &v) {
  std::sort(v.begin(), v.end());
}
```

- And let's say your mode function in stats.cpp uses sort:

```
double mode(vector<double> v) {
  assert(!v.empty());
  sort(v);
  //...
}
```

- If the staff changes the implementation of sort(), do you need to change your mode function? No!

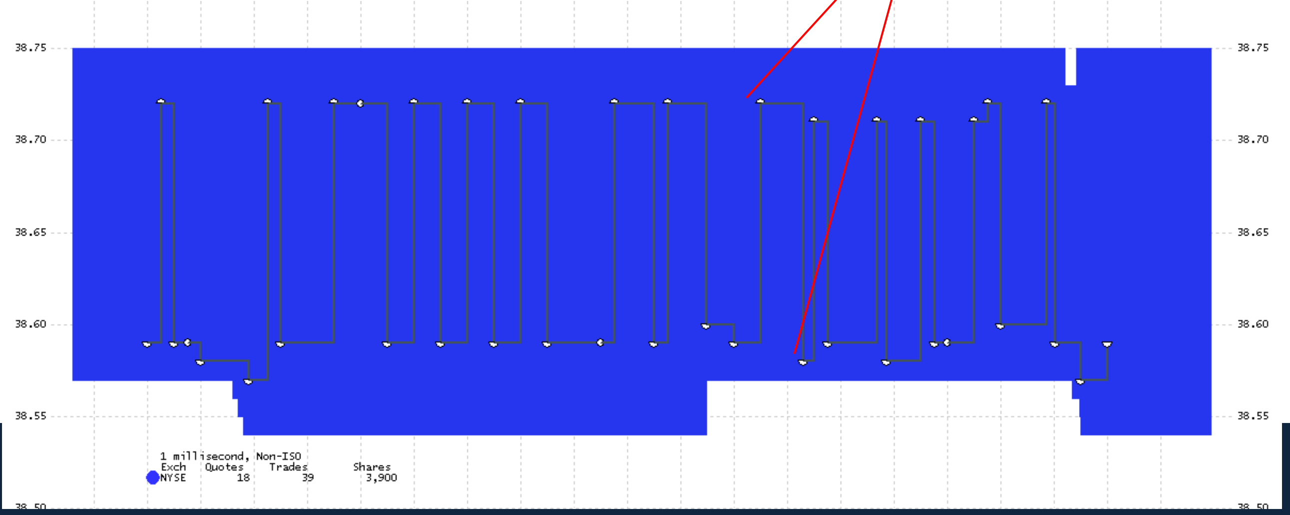
Agenda

- **Testing and debugging**
- Pointers

But I wrote it correctly!

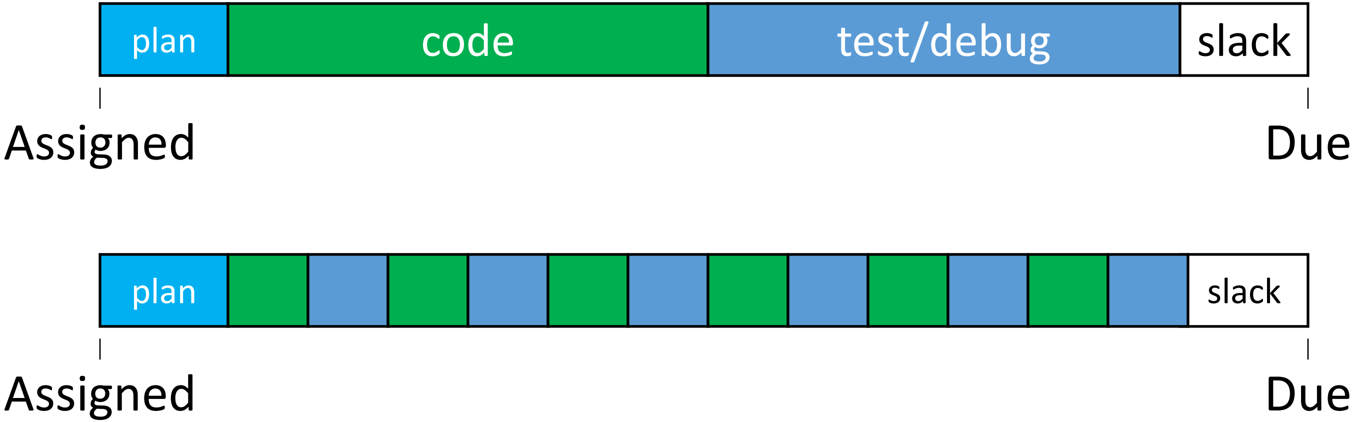
- Knight Capital Group: high-speed stock trading
  - Goal: buy low, sell high
- August 1, 2012: upgraded software algorithm
  - Bug: buy high, sell low!
- Cost $440M

trades: buy high then sell low



Motivation for Testing

- Super-linear relationship between amount of code and amount of bugs
  - More functionality, but also more connections between components



Projects in this Class

- Most of the test cases for projects are hidden!
  - You won't know what they are checking for
- You need to right good tests to convince **yourself** that it's going to work correctly
- For later projects (2-5), you will be graded on how effective your tests are at exposing possible bugs

Types of Testing

- Unit testing
  - One piece at a time (e.g., a function)
  - Find and fix bugs early! This saves you time!
    - Test smaller, less complex, easier to understand units.
    - You just wrote the code – so it's easier to debug.
- System testing
  - Entire project (code base)
  - Do this after unit testing
- Regression testing
  - Automatically run all unit and system tests after a code change

Unit tests

we don't care run wrong type of data

What's a good unit test for mode?

you don't need to care if someone break the requires

~~A)~~ mode({}) == 0
~~B)~~ mode({"text"}) == CRASH
C) mode({2,2,3}) == 2
D) mode({3,2,2,3}) == 2

- Consider test cases for the mode function from project 1...

```
// REQUIRES: v is not empty
// EFFECTS:  Returns the mode of the numbers in v.
//           http://en.Wikipedia.org/wiki/Mode_(statistics)
double mode(std::vector<double> v);
```

Simple	{2, 1, 2} → 2
(Edge) Special	{3, 2, 2, 3} → 2
Stress	{2, 2, 2, 2, 3, 3, 3, 3, 3, 3} (in 281 maybe)

Example – Unit Tests

- Let's take a look at some unit tests for project 1.

```
void test_mean_basic() {
  std::vector<double> data = {1, 2, 3};
  double expected = 2;
  double actual = mean(data);
  assert(actual == expected);
  ...
}

int main() {
  test_mean_basic();
  test_mean_edge();
  test_median_basic();
 ...
}
```

If this fails, we need to <u>debug</u> the implementation of `mean`.

The Small Scope Hypothesis

- *Thorough testing with "small" test cases is sufficient to find most bugs within a system.*

- Think about what makes two test cases meaningfully different for the function's behavior.
  - Beyond a small size, just making test cases bigger doesn't make them meaningfully different.
  - Testing with {1, 1, 2, 2, 2} is just as good as testing with {1, 1, 2, 2, 2, 2} or {1, 1, 2, 2, 2, 2, 2}.

!!WARNING!!

Important!

- Your P1 functions must work as specified WHEN RUN BY THEMSELVES
  - Just because your "summarize" function generates the correct output, doesn't mean the individual functions are correct
  - Unit test each of your functions individually to verify they work for all valid input

Regression test

- A regression test runs all unit and system tests automatically
- Make a change to your code? Quickly run a regression test and see if you broke anything
- We'll use a Makefile to run our regression test

Makefile

- Code provided in P1 Makefile for regression test

```
# Makefile
# Run regression test
test: main.exe stats_tests.exe stats_public_test.exe

./stats_public_test.exe
./stats_tests.exe
./main.exe < main_test.in > main_test.out
diff main_test.out main_test.out.correct
```

Unit tests

System test

- Run regression test at the command line:
- $ make test

Test driven development

- Write your tests first!
  - Encourages a high level understanding of the spec before implementing
  - Prevents "tunnel vision" of what tests to write
    - If you write tests after implementing code, you'll only consider the bugs you've already put effort into preventing
- Write implementations and test/debug until it passes ALL of your tests

Debugging mean

- The essential nature of debugging is to figure out precisely where your program goes wrong.
- We can narrow down where the problem is by observing the state of the program at key points.

```
double mean(vector<double> v) {
  double s = sum(v);
  //set breakpoint here to observe s

  double c = count(v);
  //set breakpoint here to observe c

  return s / c;
}
```

Think of debugging as hypothesis testing.
For example, this line tests the hypothesis "something is wrong with the sum function".

Using print statements can be kind of clunky. The setup tutorial shows you how to use a debugger

Poll Question

Which do YOU this is / will be hardest? (No wrong answer)

A) Implementing a spec
B) Writing tests
C) Debugging
D) All roughly the same

Agenda

- Testing and debugging
- **Pointers**

Pointers Motivation

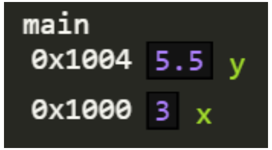
- Reference variables allowed us to do more things than default ones
  - Allows functions to modify objects created outside the scope they were created
- However, references can sometimes by clunky
  - We can't rebind reference variables to new objects
- Pointers are more flexible

```cpp
void swap(int& x,int& y) {
  int temp = x;
  x = y;
  y = temp;
}

int main() {
  int a = 3;
  int b = 7;
  swap(a, b);
  // a and b get modified
}
```

Recap: Addresses

- Every object lives at some address in memory
    - This is determined by the compiler. You don’t really have any control over it.
- You can get the address of an object using the ‘&’ operator




```
int main() {
  int x = 3;
  double y = 5.5;
  cout << &x << endl; // prints 0x1000
  cout << &y << endl; // prints 0x1004
}
```

Addresses usually printed in "hexadecimal notation"

You don't need to understand for this class

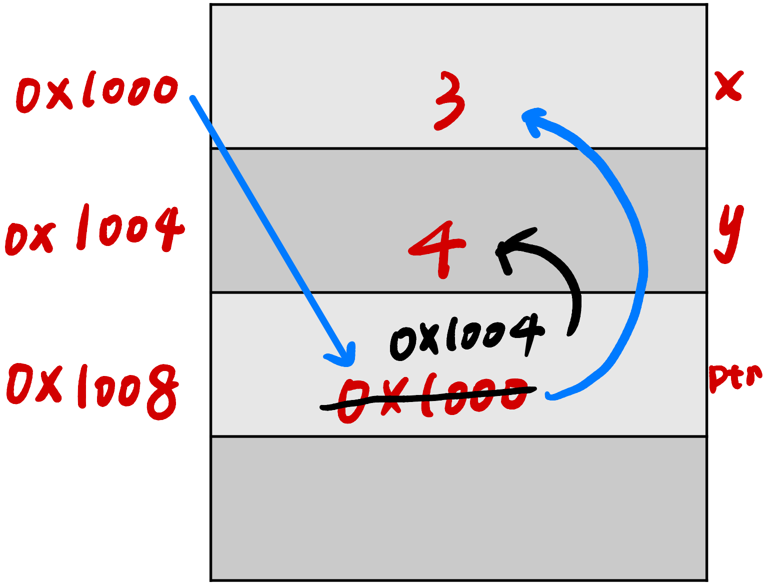
Pointers

- We can also create objects to store addresses. These are called pointers.
- To declare a pointer variable, affix the * symbol to the right of the data type

```
int main() {
  int x = 3;
  int y = 4;
  int* ptr = &x;
  cout << ptr << endl; // prints 0x1000
  ptr = &y; // assign a new address to ptr
  cout << ptr << endl; // prints 0x1004
}
```

Unlike reference variables, we can reassign pointers

Memory diagram



Dereferencing Pointers

- Once you have a pointer, you can "dereference" (i.e. follow that pointer and get the object it's looking at) via the '*' operator

```
int  x = 3;
int* p;
p = &x;
cout << *p; // prints 3 (not address)
(*p)++;
cout << *p; // prints 4
cout <<  x; // also prints 4
```

What is the data type of &*p?

A) int
B) int*
C) int&
D) int*&

Pass by Pointer

What should we pass into "swap_p"?

A) a, b
B) *a, *b
C) &a, &b

- Pointers give us an alternative to "passing by reference"

```
void swap(int& x,int& y) {
  int temp = x;
  x = y;
  y = temp;
}

int main() {
  int a = 3;
  int b = 7;
  swap(a, b);
}
```

```
void swap_p(int* x,int* y) {
  int temp = *x;
  *x = *y;
  *y = temp;
}

int main() {
  int a = 3;
  int b = 7;
  swap_p( ?? );
}
```

Pointer Details

- There is a separate pointer type for each kind of thing you could point to, and you can't mix them.

```
int main() {
  int x = 3;
  double y = 4;
  int* ptr1 = &x;
  double* ptr2 = &y;
  ptr2 = &x; // compiler error!
}
```

double int

data type miss match

Using Pointers in Expressions

- Dealing with pointer expressions can be confusing
- It can be helpful to keep track of what data type each object is
  - Using ‘&’ next to an object yields an object with an extra ‘*’ in the data type
  - Using ‘*’ next to a pointer object yields an object with one fewer ‘*’ in the data type

```
int  x = 3;

int* p;
p = &x;

int  y;
y = *p;
```

Why Pointers?

- Main tool which lets you do more in 280 vs. 183/101

- Where we'll get to: can create places in memory with no name, need to use pointers

Note on Notation

- I've been declaring pointer types this way

```
int* ptr1 = &x; // space after *
```

- But you can also do it this way

```
int *ptr1 = &x; // space before *
```

- The second one is more common in practice

Null and Uninitialized Pointers

- A null pointer has value 0x0 (i.e. it points to address 0)
  - No objects are allowed to live at address 0.
  - A null pointer is interpreted as "not pointing to anything".
  - Dereference a null pointer → undefined behavior (usually a runtime error).

```
int main() {
  int *ptr = nullptr;
  cout << ptr << endl;
  // prints 0
  cout << *ptr << endl;
  // probably crashes
}
```

Exercise: Pointers 4

resolve the problem

- Find the file "L03.4_pointer_mischief" on Lobster.

&

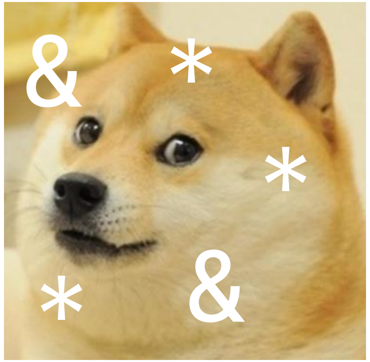
```
int * getAddress(int x) {
  return &x; // It's a trap!
}
void printAnInt(int someInt) {
  cout << someInt << endl;
}
int main() {
  int a = 3;
  int *ptr = getAddress(a);
  printAnInt(42);
  cout << *ptr << endl;
}
```

Why is it a trap?

A) Can't return pointers from functions
B) anInt became a reference to x
C) The lifetime of the parameter x ended before ptr was used
D) ptr became uninitialized when printAnInt was called

The address not change but the value located on that address changed

So Many * and &



- Used to specify a type…
  - * means it’s a pointer
  - & means it’s a reference

```
int* ptr;
```

```
int& ref;
```

- Used as an operator in an expression…
  - * means get object at an address

```
cout << *ptr << endl;
```

  - & means take address of an object

```
cout << &x << endl;
```

References vs. Pointers

References	Pointers
int& x	int* x
An alias for an object	Stores address of an object
Cannot rebind to another object	**Can** change where it points
Cannot refer to NULL (safer)	**Can** point to NULL (trickier)

```
int main() {
  int x = 3;
  int& y =  x;
  int* z = &x;
}
```

What can you do with pointers?

- Work with objects indirectly.
  - “Simulate” reference semantics.
  - Use objects across different scopes.
  - Enable subtype polymorphism.[1]
  - Keep track of objects in dynamic memory.[1]

[1] We’ll look at these later in the course.

Next Time

- Arrays
  - Probably more depth than what you've seen before
  - They have a deep relationship with pointers
- Lingering questions / feedback? I'll include an anonymous form at the end of every lecture: https://bit.ly/3oXr4Ah